(4)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: none	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Voyeur: Graphical Views of Parallel Programs		5. TYPE OF REPORT & PERIOD COVERED: Technical Report
		6. PERFORMING ORG. REPORT NUMBER: 88-04-03
7. AUTHOR(s): David Socha, Mary Bailey, David Notkin		8. CONTRACT OR GRANT NUMBER(s): N000 14-86-K-0264
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Washington, Department of Computer Science, Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Arlington, VA 22217		12. REPORT DATE: April 1988 13. NUMBER OF PAGES: 10
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

debugging, parallel programming program visualization, parallel debugging, monitoring

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Voyeur is a prototype system that facilitates the construction of application-specific, visual views of parallel programs. These views range from textual views showing the contents of variables to graphical maps of the state of the computational domain of the program. These views have been instrumental in quickly detecting bugs that would have been difficult to detect otherwise.

Keywords: parallel debugging, monitoring (KC)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Voyeur: Graphical Views of Parallel Programs

David Socha, Mary Bailey and David Notkin
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

TR 88-04-03
April 1988

Voyeur is a prototype system that facilitates the construction of application-specific, visual views of parallel programs. These views range from textual views showing the contents of variables to graphical maps of the state of the computational domain of the program. These views have been instrumental in quickly detecting bugs that would have been difficult to detect otherwise.

This research funded in part by the Office of Naval Research contract N00014-86-K-0264, National Science Foundation Grant CCR-8416878 and the Air Force Office of Scientific Research Contract 88-0023.

This paper will appear in the *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* held in Madison, WI, May 1988.

Voyeur: Graphical Views of Parallel Programs*

David Socha
Mary L. Bailey
David Notkin

Department of Computer Science, FR-35
University of Washington,
Seattle, WA 98195

Abstract

Voyeur is a prototype system that facilitates the construction of application-specific, visual views of parallel programs. These views range from textual views showing the contents of variables to graphical maps of the state of the computational domain of the program. These views have been instrumental in quickly detecting bugs that would have been difficult to detect otherwise.

1 Introduction

Debugging parallel programs is generally more challenging than debugging sequential programs. The added challenge comes in two ways. First, management of data and control may be hard, since parallel architectures are more complicated. Second, there is often an increased gap between a programmer's mental conception of a problem and the realization of a parallel program that solves that problem. Voyeur addresses this second point by allowing the parallel programmer to easily construct application-specific, visual views of parallel programs.

The original motivation for Voyeur was to improve the trace facility of Poker [16]. Three objectives characterize our initial prototype effort.

- The Voyeur prototype supports graphical visualizations (or *views*) of parallel program executions. All debuggers permit program data to be displayed in some form; Voyeur concentrates on views that are close to the programmer's mental model of a problem.
- The Voyeur prototype simplifies the task of building views. If these costs are too great, programmers will be discouraged from constructing views appropriate for a given program.
- The Voyeur prototype simplifies the task of learning how to build views. Since each parallel program might require its own view or views, it must be feasible for all parallel programmers, not just specially trained experts, to construct views effectively.

Voyeur provides a language-independent and system-independent mechanism for constructing views of parallel programs. In particular, Voyeur is separated from the data gathering process; views can be driven by a parallel architecture, by a simulator of a parallel architecture, or by a file containing trace information. To date, we have constructed Voyeur views for (1) MIMD, nonshared memory programs written in Poker and executed on a simulator, (2) a shared memory, multi-threaded program written in PRESTO[1] [1] and executed on a Sequent multiprocessor, and (3) a sequential Fortran program executed on a MicroVAX-2. This last example shows that Voyeur, while motivated by the complexity of debugging parallel Poker programs, is attractive for debugging complex sequential programs as well.

This paper describes Voyeur's structure and its use in monitoring and debugging parallel programs.

*This research funded in part by Office of Naval Research Contract N00014-86-K-0264, National Science Foundation Grant CCR-8416878, and Air Force Office of Scientific Research Contract 88-0023.

[1] Essentially C++ extended with a thread class.

Voyeur supports a hierarchy of views ranging from fairly abstract views showing images of a computational domain to language- and architecture-specific views showing just the state of program variables. Each of these views is useful for specific applications or for specific steps in monitoring and debugging. The hierarchy of views is used to reduce the cost and complexity of building views.

A Debugging Model. Parallel debugging includes five types of activity: executing the program (on a parallel architecture or on a simulator); gathering data from the program; processing the data; controlling the program's execution; and modifying the program's state. Each of these activities is complex, and there are significant interactions among them. For a debugging system to be effective each of these activities must be designed in the context of the others.

Much of the current work on parallel debugging falls into the activities of gathering the data and visualizing the data (the form of processing the data in which we are most interested).

There are several major problems associated with gathering data from parallel programs. One, nonshared memory architectures may make it difficult to access the desired data. Two, there is often a huge volume of data. Three, the non-determinism inherent in parallelism adversely affects the reproducibility of many parallel programs and their errors.

Once the data is gathered, it can be processed in one of several ways. *Program analysis* attempts to detect patterns in the program. Static (compile-time) program analysis can detect patterns in the program's description. Dynamic (run-time) program analysis detects patterns in the program's behavior over time. *Regression analysis* detects variations in successive executions. If one of the executions is a stable, correct execution, variance from that execution indicates possible bugs. *Monitoring* and *visualization* display information about the executing program. The programmer uses this information to detect errors, understand the program or its problem area, or document a program or problem area. The distinction between program visualization and program monitoring is vague, but monitoring is usually restricted to read-only views of the program or system while visualization extends to interactive images.

Related Work. Instant Replay [12] gathers just enough information to guarantee that the order in which the parallel components interact will be the same on replay, even in the presense of debugging code added for the replay. This minimizes the data gathered, and time spent collecting the data. Since Voyeur focuses on the processing phase, not the gathering phase, these approaches are orthogonal.

PECAN [13] displays multiple visualizations of sequential programs. PECAN views–such as flowcharts and symbol tables–are defined as an inherent part of the system. Hence, every program must be debugged using the same views. This contrasts with Voyeur, which encourages the construction of views specific to each program.

Several parallel programming tools use the structure of the process interconnection to display the run-time behavior of parallel programs. Poker's [16] trace view displays processor variables in boxes connected by communication links. SDEF [8] provides a similar view for systolic algorithms, and includes some information about messages and whether a process is waiting for a message to arrive. Belvedere [11] displays interprocess message activity on the interconnection graph. Belvedere also analyses the message traffic in an attempt to compensate for the asynchronous behavior of the parallel program: events of the same logical system-wide event, such as when all processes send a value in a common direction, are shown together even if they did not occur at exactly the same time.

2 Using Voyeur

We have built four Voyeur views that we have used to debug Poker [16] programs. The variety and complexity of some of the bugs we have found is interesting in itself. This section illustrates the use of Voyeur views by showing how they helped detect and locate bugs found in example parallel programs.

The main example parallel program was designed as part of an investigation into ways to dynamically balance the work load for nonshared memory algorithms. In this case, the algorithm is a simulation of sharks and fishes moving in a world composed of a two-dimensional grid of points [6]. Sharks and fishes inhabit points, one animal per point. Fish may move only into vacant points that are vertically or horizontally adjacent. If possible, sharks move to an adjacent point containing (and thus eating) a fish; otherwise, sharks move to an adjacent, vacant point. If there is a selection of available points, one is chosen randomly. Both species occasionally give birth, with the baby

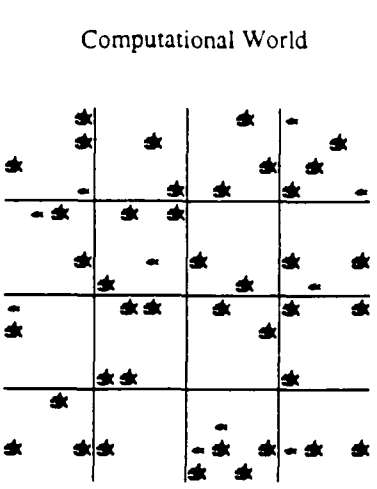

Figure 1: The World of Sharks and Fishes.

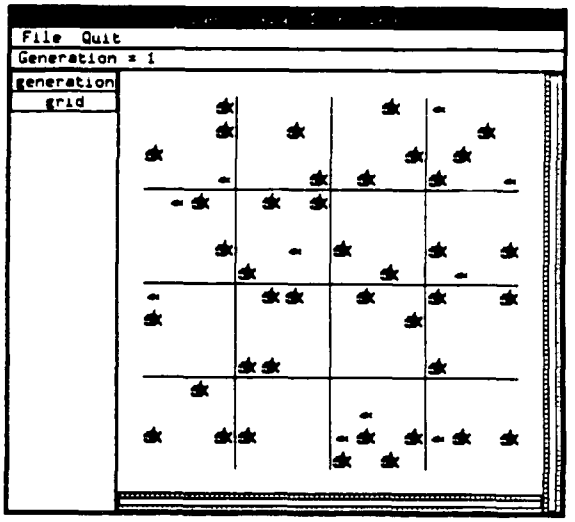

Figure 2: An icon view of a Poker program of the sharks and fishes algorithm.

staying in the place the parent vacates. Fish never starve (there is an infinite supply of plankton), but sharks do if left unfed long enough. To simplify the algorithm, the sharks and fish move in an alternating pattern: first all of the fish, then all of the sharks. The algorithm must keep sharks and fishes balanced among the processors.

We wrote a Poker program to implement the sharks and fishes algorithm. Poker is a programming language for nonshared memory, MIMD algorithms, so we allocated the world's points to 16 processors (PEs) connected in a grid; each processor controls a 4x4 square of the grid (see Figure 1).

The debugging facilities available in the Poker programming environment were insufficient to easily detect several bugs in this program, especially bugs related to randomness, to inter-PE communication, and to globally complex situations. Poker's trace view displayed only a small amount of textual information for each processor. This was insufficient to trace the 16 points in each processor.

Before Voyeur was built, we had to depend on a sequential debugger (`dbx`) to trace the execution of individual PEs executing under the Poker simulator. Unfortunately, the light-weight processes used our simulator confuse `dbx`, so we could set breakpoints for only one PE. Hence, we followed one PE to see if the fish moved correctly. During the simulation, the first fish and then the second fish moved into the vacant position to its west. Since there were only two fish in this PE, everything seemed plausible. However, when we ran the program using the Voyeur icon view, which graphically represents the two-dimensional world of sharks and fishes (see Figure 2), we saw *every* fish in *every* PE "randomly" move west. Fixing the bug in the use of the random number generator was simple, and global information made finding the bug easier. The view focuses on the essence of the problem–where the fish were and where they moved to–without wading through the actual instructions used to move the fish.

Later in the debugging process we discovered another bug. While watching the icon view, we noticed that some of the fish on the east side of a PE jumped, in one move, all of the way across to the west side, and vice versa. It turned out that one place in the program had the constants `east` and `west` reversed. Again, local knowledge might not have shown the error since only some of the fish on the either the east or west boundaries jumped. Yet the global pictorial view made it easy to identify that something was wrong and gave us information about the nature of the bug: that it had something to do with the east and west edges, but not north and south.

In another case, we discovered a bug in the code that exchanged data between PEs sharing an edge. The first clue was that some sharks were not eating fish that were adjacent but not in the same PE. When we defined more events to show what each PE thought was next to it (see Figure 3; a line next to a fish or shark indicates that the adjacent PE knows the

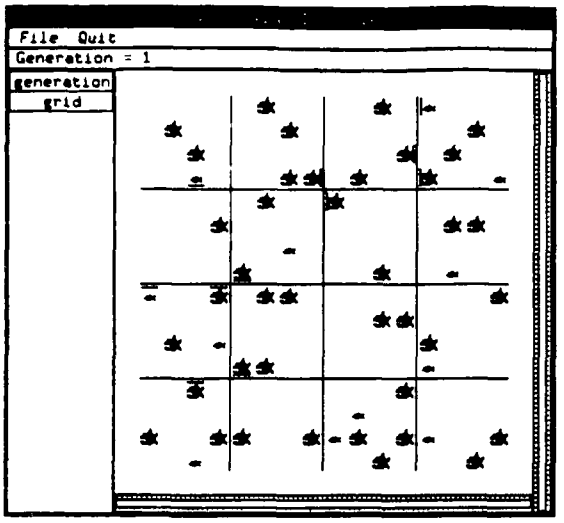

Figure 3: Using an icon view to show knowledge of adjacent PEs in a Poker program of the sharks and fishes algorithm.

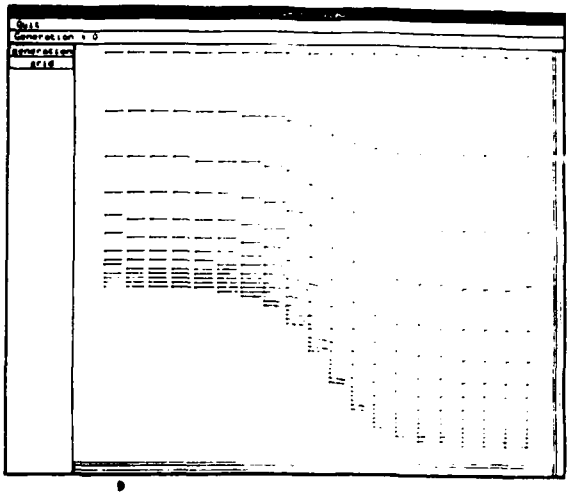

Figure 5: A vector view of a Fortran program calculating air flow through a nozzle.

fish or shark is there), we discovered that the PEs in the lower right section were not correctly transmitting their edge information. We had inadvertently replaced a manifest constant describing the width of a PE's area with a variable, and changed the local x and y values from having values relative to the upper left corner of each PE to having values relative to the upper left corner of the entire grid. However, a few constants had remained, so the coordinates of the points other than those along the top or along the left were incorrect.

The user interface of the icon view has a structure shared by all Voyeur views. It has a title bar, a set of menus, a status area, a set of buttons on the left, and a scrollable drawing area. In general, menus are used to change parameters of the view, such as appearance and connections to an executing program or its trace file. The status area displays interesting, fairly static information about what is displayed in the view's drawing region. The buttons give commands to the view or commands to control the execution of the program.

The icon view's drawing field displays 16x16 pixel icons corresponding to the occupants of each point. Actually, each type of icon (shark, fish, etc.) occupies its own plane; the view can combine these planes in a fashion analogous to overlaying overhead transparencies. Thus, if a fish and a shark happen to occupy the same point, the view would display both icons. A view can display erroneous and unexpected situations in a robust way.

The vector view was developed for a colleague in Applied Math who was translating the SIMPLE code [5] *into* Poker [9]. His simulation was failing, but he didn't know whether it was an algorithmic problem or a numerical problem resulting from the sparseness of the points in his 3-dimensional space. A quick modification of the icon view produced the vector view. Plotting the vectors at each of his 16 points, he quickly saw vectors crossing just before his simulation blew up, indicating numerical instability instead of algorithmic errors. Figure 4 shows the corrected program which still displays a numerical instability: a point in generation 5 starts moving non-radially and the simulation goes wild in generation 7. We also used this view to look at the results of a Fortran program simulating the flow of air through a nozzle (see Figure 5).

Two other views provide a closer look at actual program state. One is a more flexible version of Poker's trace view (see Figure 6). The image is structured around the communication structure of a nonshared memory parallel program. Boxes represent processors connected by communication links to other processors. Each box contains the name of the process, its position (i,j) on a grid of processors, and current values of variables within the processor. This view is generally useful for examining the values of variables during execution.

A similar view is the linked-list view. This view

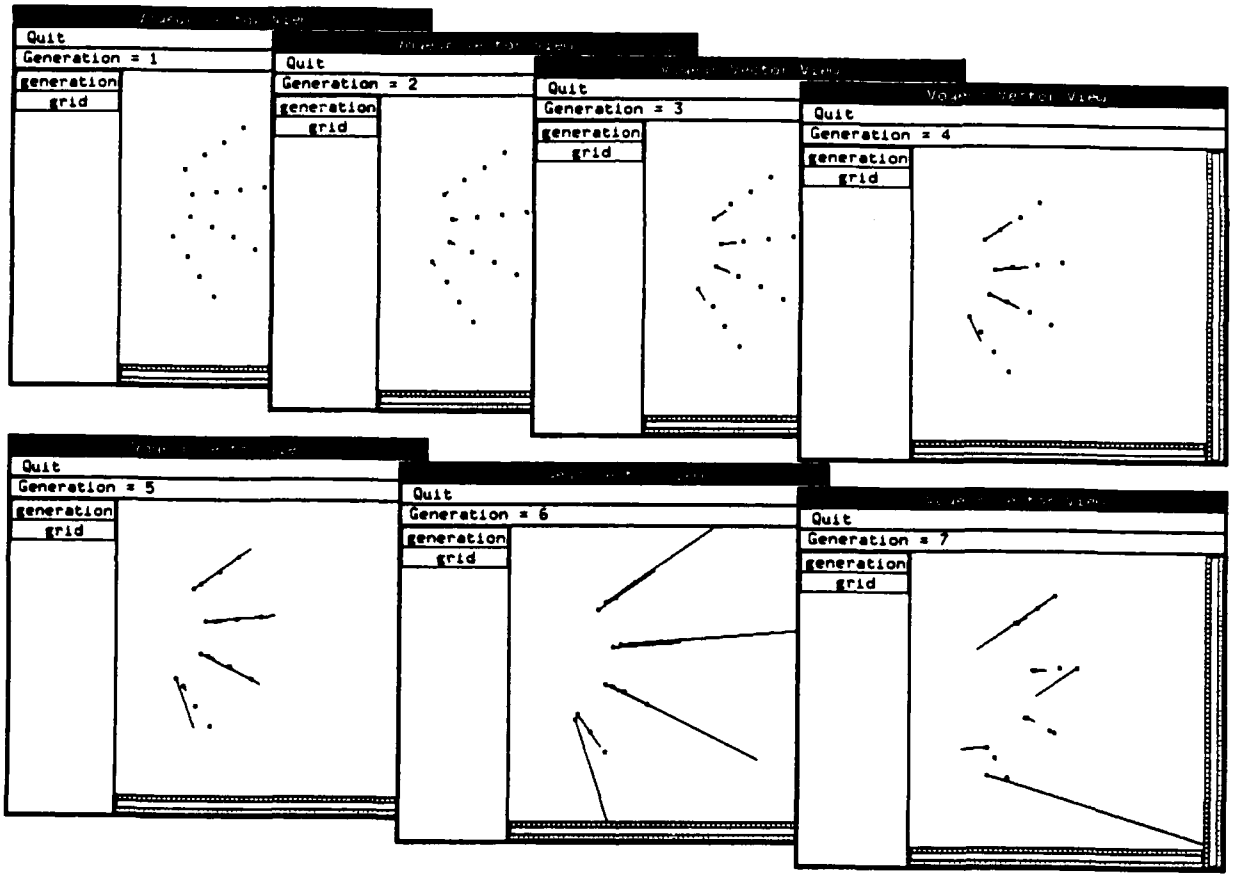

Figure 4: Using a vector view to detect numerical instabilities in a Poker program of the SIMPLE algorithm.

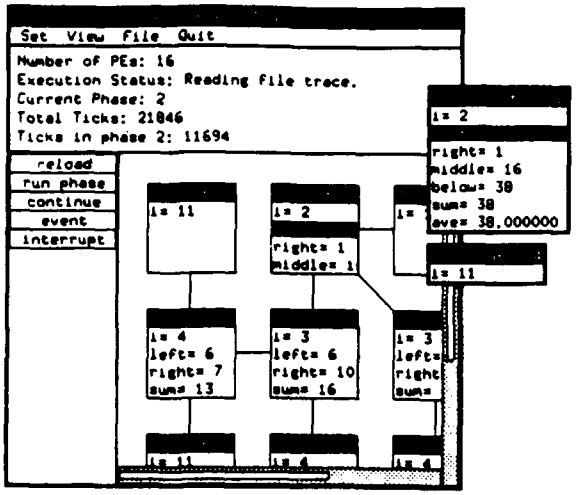

Figure 6: A enhanced version of Poker's trace view.

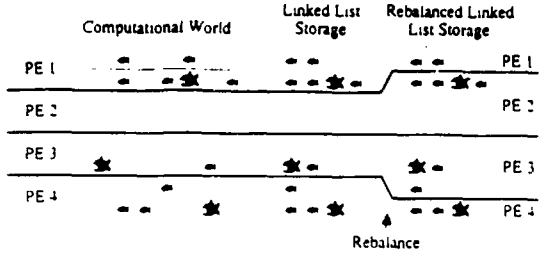

Figure 7: Rebalancing the sharks and fishes algorithm by moving rows of information between processors.

was motivated by a different allocation strategy for the sharks and fishes simulation. Instead of dividing the world grid into a coarser grid of processors, we divided it by rows. The PEs are connected in a line, instead of a grid, with each processor controlling zero or more adjacent rows (see Figure 7). As the population of sharks and fishes migrate, the simulation can automatically rebalance the load by sliding rows along the line of processors. The only constraint is that the rows maintain their order within the line of processors.

The linked-list view (see Figure 8) has two types of boxes. On top is a row of boxes corresponding to the line of PEs. Below each PE box is a list of the rows (slices) in that PE. During rebalancing the programmer can watch the slices being passed along the line of PEs.

The first use of the linked-list view detected a bug in the linked list portion of sharks and fishes. When we deleted a slice from a PE's linked list we had forgotten to update the pointers to that no-longer-existing slice. This bug was immediately visible in the linked-list view, yet had gone undetected during debugging with the Voyeur trace view. The linked-list view also allowed us to easily verify that the slices were moving correctly from one PE to the another during the rebalance phase of the program.

This view illustrates one of the philosophies underlying Voyeur: where appropriate, present the state of the information within the program, as well as synthesized information. The connecting edges do *not* necessarily contain the same information as the pointer values, since an incorrectly written Poker program could enter a state that was unexpected by the Voyeur view. The linked-list view shows the actual value of the left, right, and self pointers as well as the synthesized links connecting slices in order to aid the programmer in detecting expected *and* unexpected types of errors.

3 Voyeur System Structure

Figure 9 shows the structure of the prototype. Boxes with square corners are heavy-weight processes. Boxes with rounded corners are modules. Messages from the user filter down to change the form of the view or to request more simulation data. Messages from the simulator filter up to change the state shown by the view. The Voyeur prototype is written in C and uses X windows [14].

Voyeur uses an adapter-modeler-renderer structure similar to BALSA [4]. BALSA is a recent and impressively complete animation system used to build a large repertoire of algorithm views for the electronic classroom at Brown University. The animator manually instruments the program to generate and consume "interesting" events. The output events drive one or more modelers, which in turn drive one or more renderers (or other modelers). Each modeler keeps a model of a view independently of the display engine, window package, and so on. The renderers are responsible for translating the internal model into a visual image, displaying it, and correlating input events with the displayed objects. The input events allow the program to react, in a pre-determined fashion, to user input.

BALSA allows a large degree of reusability. An algorithm's interface is defined by the events it gen-

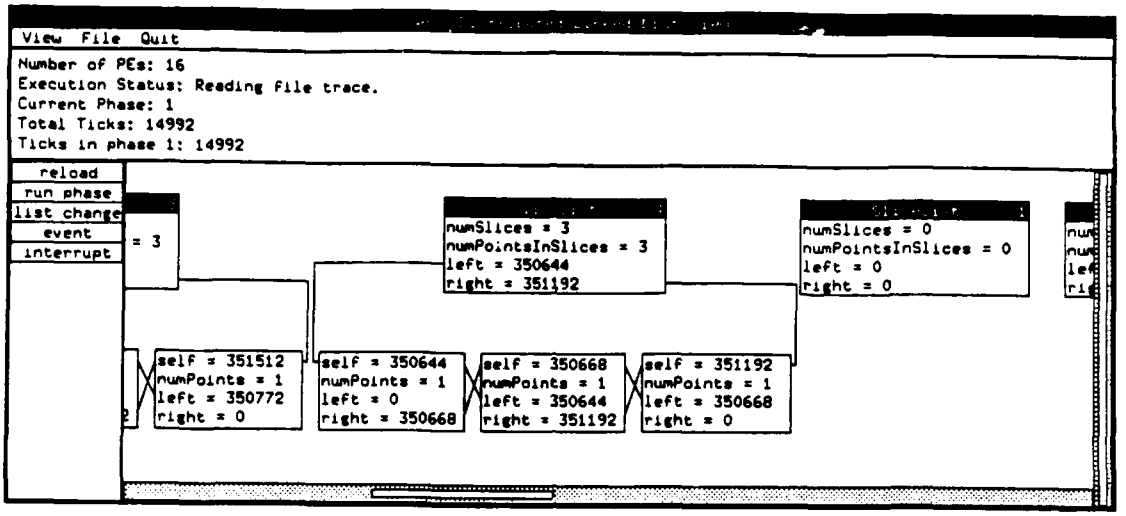

Figure 8: A view of the linked-list storage for the Poker program for the sharks and fishes algorithm.

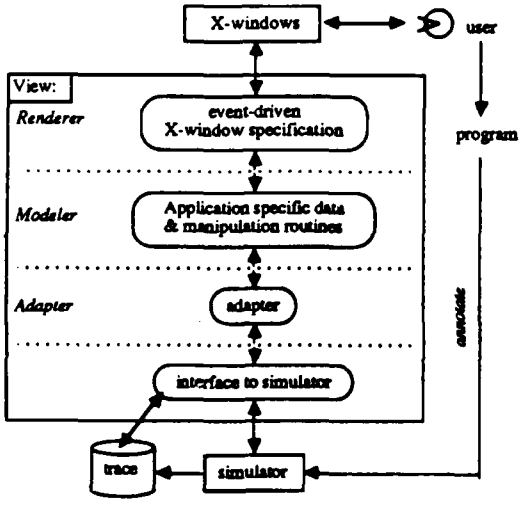

Figure 9: Voyeur's system structure.

erates and consumes. Any algorithm with the same interface may use the same set of views and input generators. Similarly, views and input generators may be attached to any program with the correct interface. Unfortunately, (1) BALSA is language specific and executes the programs within BALSA itself, and (2) BALSA does not simplify the construction of views. Still, the internal structure is a useful base from which to build.

Voyeur uses an interface to separate the program and the view. Messages from the back-end enter through the interface. The interface encapsulates knowledge of how to interact with the back-end. Thus, the view is isolated from the source of the information, whether it be from an executing program or from a trace file. The adapter, created from a simple user-supplied specification, converts the ASCII messages into calls to modeler routines. These routines modify the state of the modeler; an object maintaining a model of the information it deems interesting. Actually, the adapter filters the information and only passes on information of interest to the modeler. The renderer creates both the user interface features, such as buttons and menus, and an image of the state of the modeler. The renderer also correlates user input on the screen with screen objects to allow for visual user input. For instance, the user may select a box of the trace view by clicking in the box.

The user starts the viewer by opening a connection either to file containing traces from a previous execution or to a simulator or parallel processor, via

a pipe, for interactive viewing. The view first consumes initialization information from the simulator, such as the number of PEs. Once initialized, the user may single-step, run, or interrupt the program's execution as needed. The user may also make changes to the presentation of the view.

The back-end typically requires only minor modifications to work with read-only Voyeur views. For instance, changing the Poker simulator to allow a new type of message tagged for Voyeur took only a few minutes.

The programmer generates messages for the view by inserting instrumentation code into the parallel program. This code sends ASCII encoded messages to the view. Each message contains a prefix followed by a list of parameters. The prefix determines the type of message. The parameters are the arguments of the appropriate modeler routine called for this type of message.

For instance, to instrument the sharks and fishes program for the icon view, the programmer uses two types of messages, which look like:

```
z p 1 12 3 4 s
z g 1 12
```

The "z" indicates to Poker that this is a Voyeur message. The first message is of type "z p" and contains information about the location of a object of type "s" (a shark) at world coordinate 3,4 in PE 1 at time 12. The second message of type "z g" indicates that PE 1 has sent all messages for generation 12. The programmer uses these two messages to take snapshots of the world. The icon view consumes these snapshots one at a time with the end of each snapshot located by the generation messages.

The exact prefixes and the number, order, and type of parameters are determined by the programmer in a simple specification that is compiled to automatically create the adapter routines that decode the Voyeur messages and call the appropriate modeler routine.

4 Discussion

Voyeur simplifies view construction and modification by using a class hierarchy of views (see Figure 10). The base class contains the basic structure of the title bar, menus, status area, buttons, and scrollable drawing area. The base class also has simple mechanisms for specifying the contents of each area, except for the drawing area, and for connecting activity on a button or menu to the appropriate routine in the modeler. This user interface is built on Sx, an X window toolkit, written by John Ousterhout.

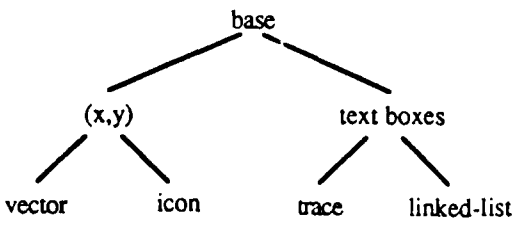

Figure 10: A class hierarchy of Voyeur views.

The vector and icon views are members of a class of x,y views. These views represent state by placing graphical elements on an x,y coordinate space filling the drawing area. The icon view uses a grid whoses points are the same size as an icon. Coordinates are given as integers. The vector view uses real-valued coordinates and can rescale its image. The similarity of these views allowed the vector view to be constructed easily from the icon view.[2] This shows how Voyeur supports easy creation of new views.

These x,y views also demonstrate the incremental flexibility of Voyeur views. Each type of information (shark, fish, edge, hydrodynamics vectors, heat vectors, etc.) is stored in a separate layer. The view can show all layers or a selected subset. This enables a user to easily add or remove information that already fits in the view's model. For instance we added the edge information in Figure 3 by simply adding a new type of message from the program, which also added a new conversion routine to the adapter. Ida [17] uses a similar, more developed idea analogous to a number of overhead transparencies which may be manipulated independently of each other and combined using a variety of graphical functions, such as *or* and *invert*. This flexibility is similar to that provided in BALSA.

Voyeur's trace view and linked-list view are examples of views derived from the text class of views. Views in this class use lines to link boxes containing textual fields. Each box may have one field per line. Text boxes are typed according the the fields they contain. For instance, the PE and slice boxes of the

[2] In reality, this took about a day since (1) we had to first abstract from the x,y view from the icon view and then build the vector view, and (2) the prototype was written in C and did not directly support object oriented programming. The next version of Voyeur is being written in C++ and will fully support the view hierarchy.

linked-list view are of different types. The user can change the order of fields within a box and the size of boxes. For instance, the slice boxes could be shrunk to display only the first line. Similarly, a trace view showing an algorithm that computes the maximum function (Figure 6) has three types: root, internal, and leaf, each of which contains a different set of traced variables (fields). Allowing the user to rearrange the order and sizes of the boxes is important when there are a large number of fields per box: the values of interest at the moment may be brought to the top so that they are still visible when using small boxes.

The text box class also allows the user to create copies of any windows and move or resize them independently from the rest of the display. This is particularly useful in two situations. First, duplicate boxes allows the user to view boxes that are too far apart in the drawing region to fit together on the screen. Second, using small boxes for the drawing region allows us to view global patterns while a few particularly interesting boxes may be copied and expanded to show a large amount of local information.

In summary, Voyeur has met its three initial objectives.

- A view programmer has full access to the graphical capabilities of X windows [14]. This meets our objective of supporting graphical, visual views.
- By using a clean and general interface to separate the viewing mechanism (front-end) from the parallel program (back-end) we have increased the reusability of views among parallel programs and among parallel systems. This meets our objective of simplifying the task of building views.
- By separating the view from the parallel program and by organizing views in an object hierarchy, we have simplified the task of learning how to build views. This meets out objective of allowing virtually all parallel programmers to develop problem- and program-specific views.

5 Future Work

There is still a large amount of work and thought needed to create a system that fulfills the goal of allowing parallel programmers to easily build application-specific, visual views of parallel programs. We are investigating several metaphors and system structures that might provide the basis for better solutions.

- *Ease of construction.* Efforts such as Ida [17], ARK [15], ThinkerToy [10], ThingLab [2, 3], and Duisberg's gestural interface for animating algorithms [7] are reducing the cost of building interfaces and the cost of learning *how to* build user interfaces.
- *Modifying program state.* Visualization work to date has not dealt with changing the state of the information (data or program state) in response to user interactions with the visualization. This form of experimental "What if?" questioning is a feature of traditional debuggers that is quite useful in hunting down the cause of many a bug. Visual debugging systems must support read and write access to the program's state.
- *Multiple views.* Visual debugging systems need to include mechanisms for coordinating multiple views of one or more programs. Different views are useful for different purposes and often complement each other.
- *Interface with back-ends* The interface to the back-ends is crucial to allow the viewing mechanism and program system enough flexibility to support the range of desired visualizations. We are investigating better ways to encapsulate the knowledge of how to interact with back-ends.
- *Synthesis.* The efforts of visual programming, user interface design, data and program visualization, and debugging need to be combined to meet the needs of an easy to use visual debugging system.

As we continue to incrementally develop Voyeur, the experience from constructing and using the views gives valuable feedback and motivation for the design of a more complete system. Voyeur also provides a valuable tool for other projects needing visual front-ends to debugging, monitoring, and data analysis systems. The reusability of the Voyeur system and its views saves time for everyone.

Acknowledgments Voyeur would be a useless effort without the parallel (and sequential) programs that have driven its creation and evolution. We thank Kevin Gates for the Poker SIMPLE program; Yeng Bun for the Fortran program simulating air flowing through a nozzle; and David Wagner for the PRESTO simulation of a message-passing torus.

References

[1] Brian Bershad, Ed D. Lazowska, and H. M. Levy. Presto: A system for object-oriented parallel programming. Technical Report 87-09-01, Department of Computer Science, University of Washington, September 1987.

[2] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Language Systems*, 3(4):353–387, October 1981.

[3] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, October 1986.

[4] M. H. Brown. *Algorithm Animation*. PhD thesis, Department of Computer Science, Brown University, April 1987.

[5] W. P. Crowley, C. P. Hendrickson, and T. L. Day. The simple code. Technical Report UCID-17715, Lawrence Livermore Laboratory, February 1987.

[6] A. K. Dewdney. Computer recreations. *Scientific American*, pages 18–24, July 1984.

[7] Robert Duisberg. Visual programming of program visualizations. Technical Report 87-20, Computer Research Laboratory, Tektronix, Inc., February 1987.

[8] B. R. Engstrom and P. R. Capello. The SDEF systolic programming system. In *Proceedings of the International Conference on Parallel Processing*, pages 645–652, St. Charles, IL, August 1987.

[9] Kevin Gates. SIMPLE, an exercise in programming in Poker. Technical Report 88-2, Department of Applied Math, University of Washington, March 1988.

[10] Steven H. Gutfreund. ManiplIcons in ThinkerToy. In *Proceedings of 1987 Object-Oriented Programming Systems, Languages, and Applications*, pages 307–317, October 1987.

[11] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proceedings of the International Conference on Parallel Processing*, pages 735–738, St. Charles, IL, August 1987.

[12] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[13] Steven P. Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.

[14] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[15] Randall B. Smith. The alternate reality kit: An animated environment for creating interactive simulation. In *1986 IEEE Computer Society Workshop on Visual Languages*, pages 99–106, June 1986.

[16] Lawrence Snyder. Parallel programming and the Poker programming environment. *IEEE Computer*, 17(7):27–36, July 1984.

[17] Robert L. Young. An object-oriented framework for interactive data graphics. *Proceedings of 1987 Object-Oriented Programming Systems, Languages, and Applications*, pages 78–90, October 1987.